Lambda Calculus - Combinators (8A)

Young Won Lim
11/13/24

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

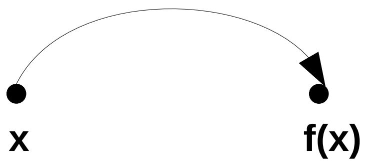
Fix point (1)

In mathematics, a **fixed point** (**fixpoint**),
also known as an **invariant point**,
is a value that does not change under a given transformation.

Specifically, for functions,
a **fixed point** is an element
that is mapped to itself by the function.

Formally, **c** is a fixed point of a function **f**
if **c** belongs to both the domain and the codomain of **f**, and

f(c) = c.



c fixed point **f(c) = c**

https://en.wikipedia.org/wiki/Fixed_point_(mathematics)

Fix point (2)

For example, if **f** is defined on the real numbers by

$f(x) = x^2 - 3x + 4$,

then 2 is a fixed point of **f**, because **f**(2) = 2.

Not all functions have fixed points: for example,

$f(x) = x + 1$, has no fixed points,

since **x** is never equal to **x** + 1 for any real number.

In graphical terms, a fixed-point **x** means

the point (**x**, **f**(**x**)) is on the line **y** = **x**, or in other words

the graph of **f** has a point in common with that line.

https://en.wikipedia.org/wiki/Fixed_point_(mathematics)

Extensionality (1)

In logic, extensionality, or extensional equality,
refers to principles that judge objects to be equal
if they have the same external properties.

It stands in contrast to the concept of intensionality,
which is concerned with whether
the internal definitions of objects are the same.

https://en.wikipedia.org/wiki/Extensionality

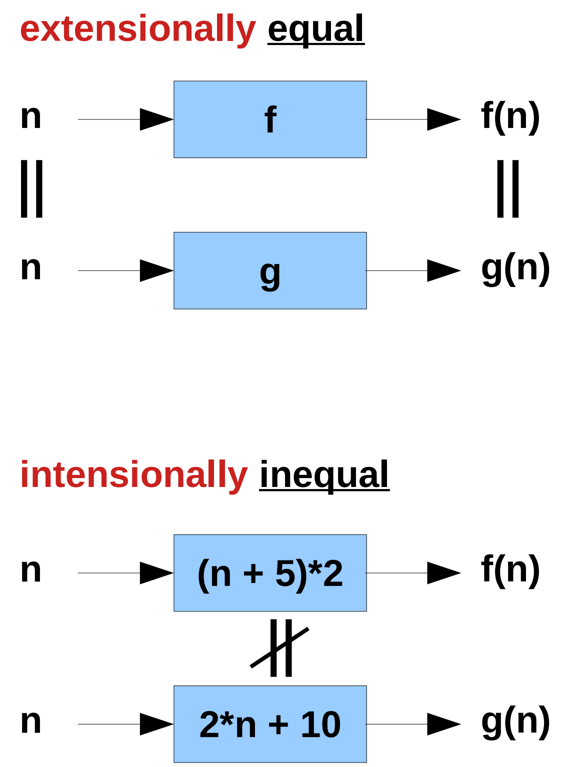
Extensionality (2)

Consider the two functions **f** and **g**
mapping from and to natural numbers,
defined as follows:

To find **f(n)**, first add **5** to **n**, then multiply by **2**. (n + 5)*2
To find **g(n)**, first multiply **n** by **2**, then add **10**. 2*n + 10

These functions are extensionally equal;
given the same input, both functions always produce the same value.

But the definitions of the functions are not equal,
and in that intensional sense the functions are not the same.



https://en.wikipedia.org/wiki/Extensionality

Extensionality (3)

Similarly, in natural language
there are many predicates (relations)
that are intensionally different
but are extensionally identical.

For example, suppose that a town has one person named Joe,
who is also the oldest person in the town.
Then, the two predicates "being called Joe",
and "being the oldest person in this town"
are intensionally distinct,
but extensionally equal
for the (current) population of this town.

https://en.wikipedia.org/wiki/Extensionality

Combinatory Logic

Combinatory logic is a notation
to eliminate the need for quantified variables in mathematical logic.

It was introduced by Moses Schönfinke and Haskell Curry,
and has more recently been used in computer science
as a theoretical model of computation
and also as a basis for the design of
functional programming languages.

It is based on combinators

without using quantified variables

theoretical model of computation

functional programming

combinators

https://en.wikipedia.org/wiki/Combinatory_logic

Combinator

combinators were introduced by Schönfinkel in 1920
with the idea of providing an analogous way
– to build up functions
– to remove any mention of variables
– particularly in predicate logic.

A combinator is a higher-order function
that uses only function application

earlier defined combinators
to define a result from its arguments.

Combinators:
define a result by its argument
without free variables

https://en.wikipedia.org/wiki/Combinatory_logic

Combinator Definitions (1)

Combinator : A lambda expression containing no free variables.

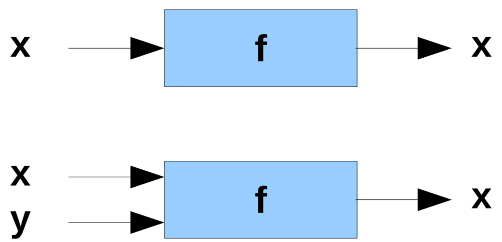
the word is usually understood more specifically
to refer to certain combinators of special importance,
in particular the following four:

I = λx . x — **Identity**

K = λx . λy . x — **Constant function**

S = λx . λy . λz . x(z)(y(z)) — **Substitution operator**

Y = λf . (λu . f(u(u))) (λu . f(u(u)))



https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/combinator

Combinator informal description (1-1)

Informally, a tree (**xy**) can be thought of

as a function **x** applied to an argument **y**.

When evaluated (i.e., when the function is "applied" to the argument),

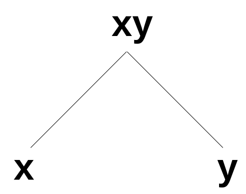
the tree "returns a value", i.e., transforms into another tree.

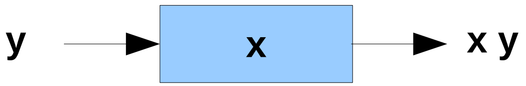
The "function", "argument" and the "value" are

either combinators or binary trees.

If they are binary trees,

they may be thought of as functions too, if needed.





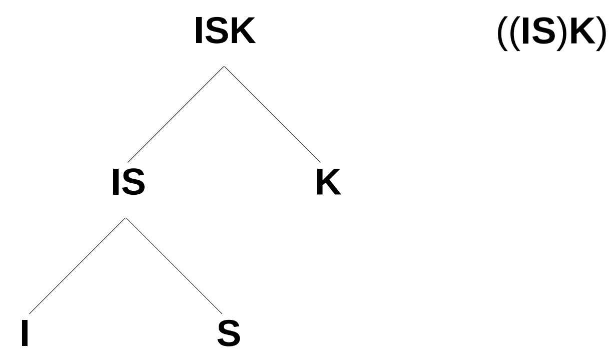
https://en.wikipedia.org/wiki/SKI_combinator_calculus

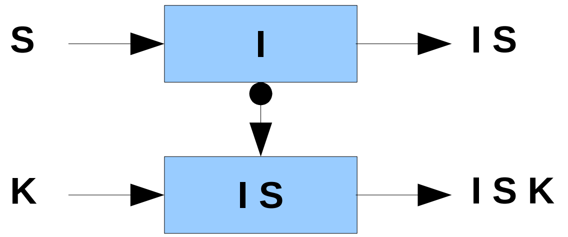
Combinator informal description (1-2)

Although the most formal representation of the objects in this system requires binary trees,
for simpler typesetting
they are often represented as parenthesized expressions,
as a shorthand for the tree they represent.

Any subtrees may be parenthesized,
but often only the right-side subtrees are parenthesized,
with left associativity implied for any unparenthesized applications.

For example, **ISK** means ((**IS**)**K**).





https://en.wikipedia.org/wiki/SKI_combinator_calculus

Combinator informal description (1-3)

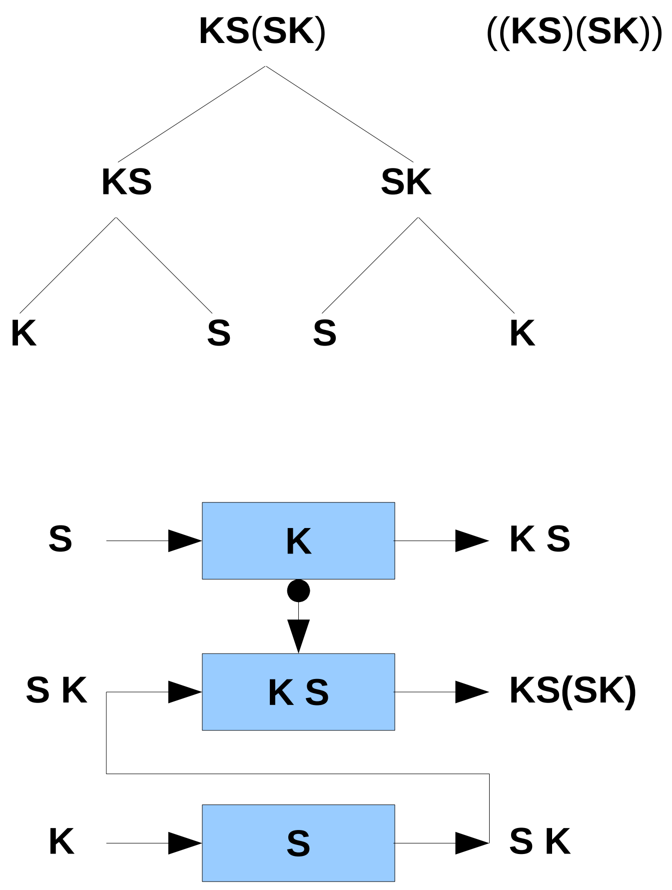
a tree whose left subtree is the tree **KS**

and whose right subtree is the tree **SK**

can be written as **KS**(**SK**).

If more explicitness is desired,

the implied parentheses can be included as well: ((**KS**)(**SK**)).



https://en.wikipedia.org/wiki/SKI_combinator_calculus

I combinator

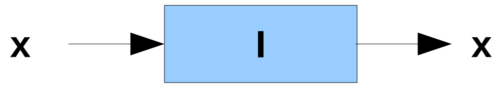
The evaluation operation is defined as follows:

x, **y**, and **z** represent expressions

made from the functions **S**, **K**, and **I**, and set values:

I returns its argument:

I x = x



https://en.wikipedia.org/wiki/SKI_combinator_calculus

Examples of Combinators (1-1)

The simplest example of a combinator is **I**, the identity combinator, defined by

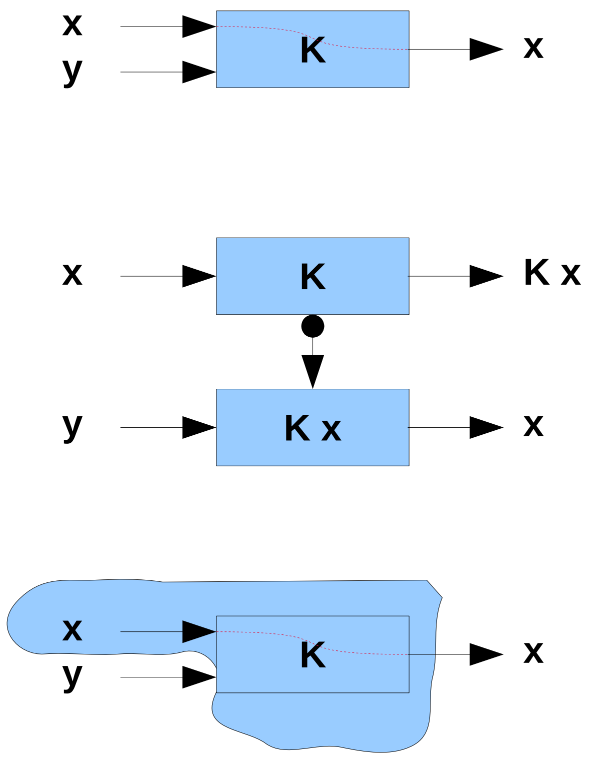
(I x) = x for all terms **x**.

https://en.wikipedia.org/wiki/Combinatory_logic

K combinator

K, when applied to any argument **x**,
yields a one-argument constant function **K x**,
which, when applied to any argument **y**, returns **x**:

K x y = x



https://en.wikipedia.org/wiki/SKI_combinator_calculus

Examples of Combinators (1-2)

Another simple combinator is **K**,

which manufactures constant functions:

(K x) is the function which, for any argument, returns **x**, so we say

((K x) y) = x for all terms **x** and **y**.

Or, following the convention for multiple application,

(K x y) = x

https://en.wikipedia.org/wiki/Combinatory_logic

S combinator

S is a substitution operator.

takes three arguments (**x y z**)

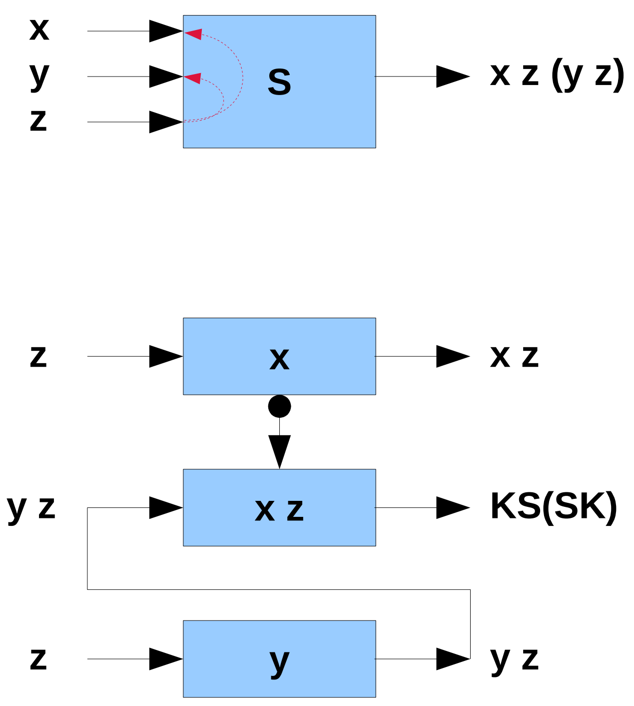
returns the result of **x z** applied to the result of **y z**

the first argument (**x**) applied to the third (**z**),

which is then applied to the result

of the second argument (**y**) applied to the third (**z**).

S x y z = x z (y z)



a function of **x z** with the argument **y z**
a function of **x** with the argument **z**
a function of **y** with the argument **z**

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Examples of Combinators (2-1)

A third combinator is **S**, which is a generalized version of application:

(S x y z) = (x z (y z))

S applies **x** to **y**
after first substituting **z** into each of them (**x** and **y**)

x is applied to **y**
inside the environment **z**.

https://en.wikipedia.org/wiki/Combinatory_logic

Examples of Combinators (2-2)

Given **S** and **K**, **I** itself is *unnecessary*,

since it can be built from the other two:

((S K K) x)

= (S K K x)

= (K x (K x))

= x

for any term **x**.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinator informal description (3-1)

SKSK evaluates to **KK**(**SK**) by the S-rule.

Then if we evaluate **KK**(**SK**), we get **K** by the K-rule.

As no further rule can be applied, the computation halts here.

For all trees **x** and all trees **y**,

SKxy will always evaluate to **y** in two steps, **Ky**(**xy**) = **y**,

so the ultimate result of evaluating **SKxy**

will always equal the result of evaluating **y**.

We say that **SKx** and **I** are "functionally equivalent" for any **x**

because they always yield the same result when applied to any **y**.

S x y z = x z (y z) S-rule

S K S K = K K (S K)

K x y = x K-rule

K K (S K) = K

S K x y = K y (x y) = y

I y = y

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Combinator informal description (3-2)

it can be shown that SKI calculus is not the minimum system
that can fully perform the computations of lambda calculus,

as all occurrences of I in any expression can be replaced
by (**SKK**) or (**SKS**) or (**SK x**) for any **x**,
and the resulting expression will yield the same result.

So the "**I**" is merely syntactic sugar.

Since **I** is optional, the system is also referred
as SK calculus or
SK combinator calculus.

S K K y = K y (K y) = y

I y = y

S K S y = K y (S y) = y

I y = y

S K x y = K y (x y) = y

I y = y

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Examples of Combinators (3-1)

Note that although **((S K K) x) = (I x)** for any **x**,
(S K K) itself is not equal to **I**.

We say the terms are extensionally equal.

Extensional equality captures the mathematical notion
of the equality of functions:

that two functions are equal
if they always produce the same results for the same arguments.

https://en.wikipedia.org/wiki/Combinatory_logic

Examples of Combinators (3-2)

In contrast, the terms themselves,

together with the reduction of primitive combinators,

capture the notion of intensional equality of functions:

that two functions are equal

only if they have identical implementations

up to the expansion of primitive combinators.

https://en.wikipedia.org/wiki/Combinatory_logic

Examples of Combinators (3-3)

There are many ways to implement an identity function;

(S K K) and **I** are among these ways.

(S K S) is yet another.

We will use the word equivalent to indicate extensional equality,

reserving equal for identical combinatorial terms.

https://en.wikipedia.org/wiki/Combinatory_logic

Examples of Combinators (4)

A more interesting combinator is

the fixed point combinator or **Y combinator**,

which can be used to implement recursion.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinator Definitions (2)

The combinators **I**, **K**, and **S** were introduced by Schönfinkel and Curry, who showed that any λ-expression can essentially be formed by combining them.

More recently combinators have been applied to the design of implementations for functional languages.

In particular **Y** (also called the paradoxical combinator) can be seen as producing fixed points, since **Y(f)** reduces to **f(Y(f))**.

I = λx . x

K = λx . λy . x

S = λx . λy . λz . x(z)(y(z))

Y = λf . (λu . f(u(u))) (λu . f(u(u)))

https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/combinator

Combinatory Logic and Lambda Calculus (1)

Lambda calculus is concerned with objects called lambda-terms, which can be represented by the following three forms of strings:

v

λv. $\mathbf{E_1}$

$\mathbf{(E_1\ E_2)}$

where **v** is a variable name drawn from a predefined infinite set of variable names, and $\mathbf{E_1}$ and $\mathbf{E_2}$ are lambda-terms.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (2)

Terms of the form **λv. E_1** are called abstractions.

The variable **v** is called the formal parameter of the abstraction,

and **E_1** is the body of the abstraction.

The term **λv. E_1** represents the function

applied to an argument,

binds the formal parameter **v** to the argument

computes the resulting value of **E_1**

returns **E_1**, with every occurrence of **v** replaced by the argument.

v

λv. E_1

(E_1 E_2)

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (3-1)

Terms of the form (**E_1 E_2**) are called **applications**.

applications model function invocation or execution:

the function represented by **E_1** is to be invoked,

with **E_2** as its argument, and the result is computed.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (3-2)

If E_1 (the applicand) is an abstraction, the term may be reduced:

E_2, the argument, may be substituted into the body of E_1

in place of the formal parameter **v** of E_1,

and the result is a new lambda term which is equivalent to the old one.

If a lambda term contains no subterms of the form $((\lambda v.\ E_1)\ E_2)$

then it cannot be reduced, and is said to be in normal form.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (4)

The motivation for this definition of reduction is
that it captures the essential behavior of all mathematical functions.

For example, consider the function
that computes the square of a number. We might write

The **square** of **x** is **x * x** (using ***** to indicate multiplication.)

x here is the formal parameter of the function.
To evaluate the **square** for a particular argument, say 3,
we insert it into the definition in place of the formal parameter:
The **square** of **3** is **3 * 3**

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (5)

To evaluate the resulting expression **3 * 3**, we would have to resort to our knowledge of multiplication and the number **3**.

Since any computation is simply a composition of
the evaluation of suitable functions
on suitable primitive arguments,

this simple substitution principle suffices
to capture the essential mechanism of computation.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (6)

Moreover, in lambda calculus, notions such as '**3**' and '*****'
can be represented without any need for externally defined
primitive operators or constants.

It is possible to identify terms in lambda calculus,
which, when suitably interpreted, behave like the number **3**
and like the multiplication operator *****, q.v. Church encoding.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (7)

Lambda calculus is known to be computationally equivalent
in power to many other plausible models for computation
(including Turing machines);

that is, any calculation that can be accomplished
in any of these other models can be expressed in lambda calculus,
and vice versa.

According to the Church-Turing thesis,
both models can express any possible computation.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (8-1)

lambda-calculus can represent any conceivable computation
using only the simple notions
of function abstraction and application
based on simple textual substitution of terms for variables.

abstraction is not even *required*.

Combinatory logic is
a model of computation equivalent to lambda calculus,
but without abstraction.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (8-2)

Combinatory logic is
a model of computation equivalent to lambda calculus,
but without abstraction.

The advantage of this is that
evaluating expressions in lambda calculus is quite complicated
because the semantics of substitution must be specified
with great care to avoid variable capture problems.

evaluating expressions in combinatory logic is much simpler,
because there is no notion of substitution.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Calculus

abstraction is the only way to manufacture functions

in the lambda calculus

Instead of abstraction,

combinatory calculus provides a limited set of primitive functions

out of which other functions may be built.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Terms (1)

A combinatory term has one of the following forms:

Syntax	Name	Description
x	Variable	A character or string representing a combinatory term.
P	Primitive function	One of the combinator symbols **I**, **K**, **S**.
(M N)	Application	Applying a function to an argument. **M** and **N** are combinatory terms.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Terms (2)

The primitive functions are combinators, or functions that,

when seen as lambda terms, contain no free variables.

To shorten the notations, a general convention is that ($\mathbf{E_1}$ $\mathbf{E_2}$ $\mathbf{E_3}$ . . . $\mathbf{E_n}$),

or even $\mathbf{E_1}$ $\mathbf{E_2}$ $\mathbf{E_3}$ . . . $\mathbf{E_n}$, denotes the term (. . . (($\mathbf{E_1}$ $\mathbf{E_2}$) $\mathbf{E_3}$) . . . $\mathbf{E_n}$) .

This is the same general convention (left-associativity)

as for multiple application in lambda calculus.

https://en.wikipedia.org/wiki/Combinatory_logic

Reductions in Combinatory Logic

In combinatory logic, each primitive combinator comes with a reduction rule of the form

$$(\mathbf{P}\ \mathbf{x}_1 \ldots \mathbf{x}_n) = \mathbf{E}$$

where **E** is a term mentioning only variables from the set $\{\mathbf{x}_1 \ldots \mathbf{x}_n\}$.

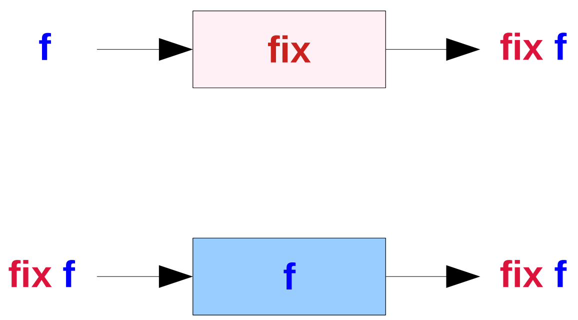
It is in this way that primitive combinators behave as functions.

https://en.wikipedia.org/wiki/Combinatory_logic

Fix-point combinator (1-1)

a **fixed-point combinator**
(or fixpoint combinator),
denoted **fix**, is a higher-order function
which takes a function **f** as argument
that returns some fixed point **(fix f)**
(a value that is mapped to itself)
of its argument function **f**, if one exists.

fix f = f (fix f),



https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (1-2)

some fixed point **(fix f)** of its argument function **f**, if one exists.

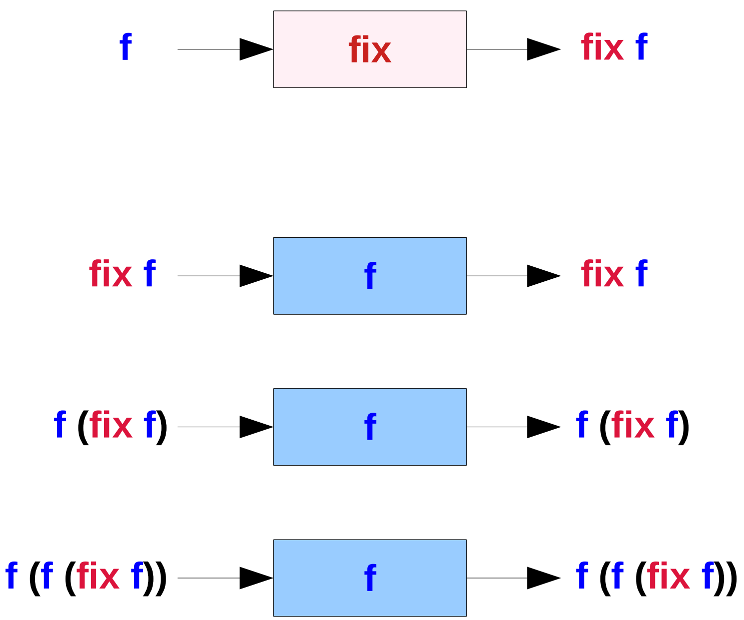
Formally, if the **function f** has one or more fixed points, then

fix f = f (fix f),

and hence, by repeated application,

fix f = f (f (… f (fix f) …))

fix f	fixed point
fix	fixed point combinator



Y = (λf. (λx.f (x x)) (λx. f (x x)))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (1111)

Every recursively defined function can be seen
as a fixed point of some suitably defined function
closing over the recursive call with an extra argument,

and therefore, using **Y**, every recursively defined function
can be expressed as a lambda expression.

In particular, we can now cleanly define
the subtraction, multiplication and comparison predicate
of natural numbers recursively.

Y = λf. (λx.f (x x)) (λx. f (x x))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Fix-point combinator (3-1)

In the classical untyped lambda calculus,
every function has a fixed point.

A particular implementation of fix is
Curry's paradoxical **combinator Y**, represented by

$$\mathbf{Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}$$

In functional programming, the **Y combinator** can be used
to formally define recursive functions in a programming language
that does not support recursion.

https://en.wikipedia.org/wiki/Fixed-point_combinator

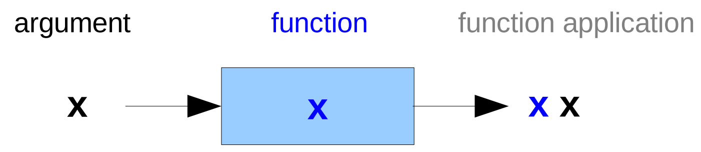
Fix-point combinator (3-1)

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Y is a function that <u>takes</u> one argument **f** and <u>returns</u> the entire expression following the first period;

$$(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

the expression $(\lambda x.\ f\ (x\ x))$ denotes a function that <u>takes</u> <u>one</u> argument **x**, thought of as a function, and <u>returns</u> the expression **f (x x)**,



https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-1)

Y = λf. (λx. f (x x)) (λx. f (x x))

the expression **(λx. f (x x))** denotes a function

that takes one argument **x**,

which is thought of as a function,

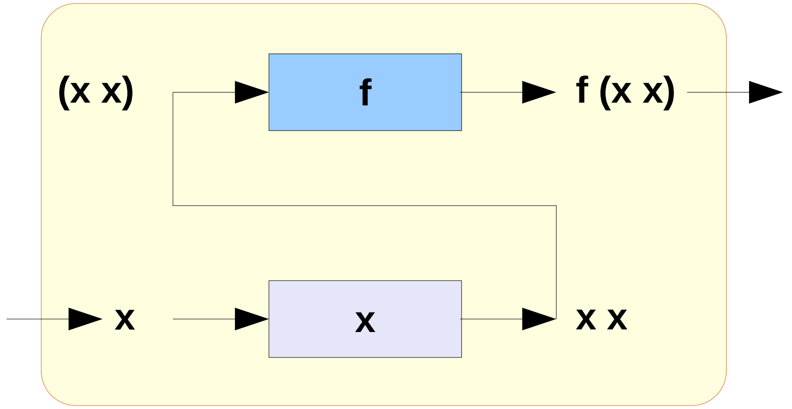
and returns the expression **f (x x)**,

where **(x x)** denotes

a function **x** applied to itself as an argument.

Juxtaposition of expressions denotes function application,

is left-associative, and has higher precedence than the period.)

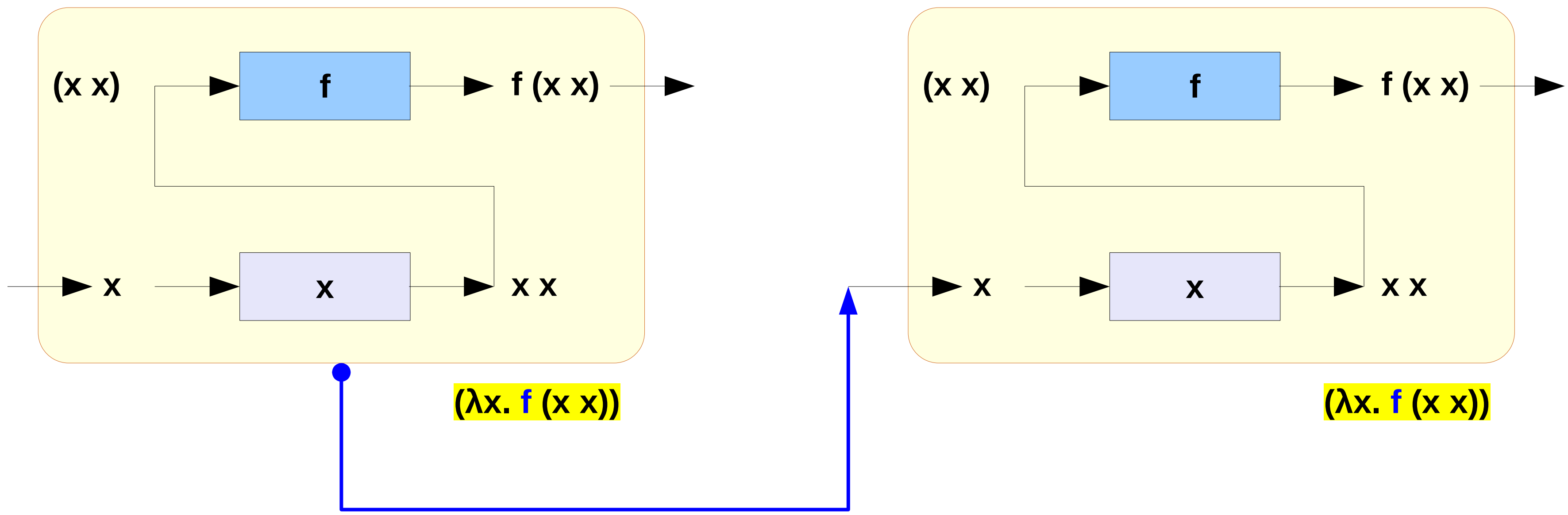


(λx. f (x x))

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-1)

Y = λf. (λx. f (x x)) (λx. f (x x))



https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-2)

The following calculation verifies that **Y g** is indeed a fixed point of the function **g** :

Y g	**= (λf. (λx.f (x x)) (λx. f (x x))) g**	by the definition of **Y**
	= (λx. g (x x)) (λx. g (x x))	by β-reduction: replacing the formal argument **f** of **Y** with the actual argument **g**
	= g ((λx. g (x x)) (λx. g (x x)))	by β-reduction: replacing the formal argument x of the first function with the actual argument **(λx. g (x x))**
	= g (Y g)	by second equality, above

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-3)

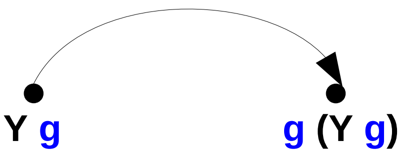
The following calculation verifies that **Y g** is indeed a fixed point of the function **g** :

Y g	**= (λf. (λx.f (x x)) (λx. f (x x))) g**	by the definition of **Y**
	= g (Y g)	by second equality, above

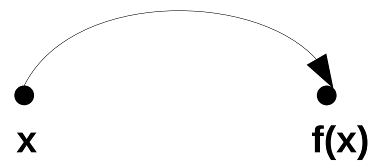
The lambda term **g (Y g)** may <u>not</u>,

in general, β-reduce to the term **(Y g)** .

However, both terms β-reduce to the same term, as shown.



Y g fixed point **g (Y g) = Y g**



c fixed point **f(c) = c**

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-2)

This combinator may be used in implementing **Curry's paradox**.

The heart of Curry's paradox is
that untyped lambda calculus is unsound as a deductive system,

and the **Y combinator** demonstrates this
by allowing an anonymous expression
to represent **zero**, or even many **values**.

This is inconsistent in mathematical logic.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-3)

Applied to a function with one variable,
the **Y combinator** usually does not terminate.

More interesting results are obtained
by applying the Y combinator to functions of two or more variables.

the additional variables may be used as a counter, or index.

the resulting function behaves like a **while** or a **for** loop
in an imperative language.

Y = (λf. (λx.f (x x)) (λx. f (x x)))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (3-3)

Used in this way, the **Y combinator** implements simple recursion.

The lambda calculus does not allow
a function to appear as a term in its own definition
as is possible in many programming languages,

but a function can be passed as an argument
to a higher-order function that applies it in a recursive manner.

Y = (λf. (λx.f (x x)) (λx. f (x x)))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (1)

The factorial function provides a good example of
how a fixed-point combinator may be used
to define recursive functions.

The standard recursive definition of the factorial function
in mathematics can be written as

$$\textbf{fact n} = \begin{cases} \textbf{1} & \textbf{if} \quad \textbf{n = 0} \\ \textbf{n fact (n-1)} & \textbf{otherwise.} \end{cases}$$

where **n** is a non-negative integer.

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (2)

If we want to implement this in lambda calculus,

- integers are represented using Church encoding,

the problem is that the lambda calculus

does not allow the name of a function ('**fact**')

to be used in the function's definition.

this problem can be circumvented

using a fixed-point combinator **fix** as follows.

fix f = f (fix f)

fix F = F (fix F),

fix f = f (fix f),

fix f fixed point

fix fixed point combinator

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (3-1)

using a fixed-point combinator **fix** as follows.

fix f = f (fix f)

fix F = F (fix F),

Let the fixed point **(fix F)** of **F** as **fact**

fact ≡ **fix F**

(fix F) = F (fix F)

(fact) = F (fact) fixed-point **fact**

(fact n) = F (fact n)

fix F = F (fix F),

fix F fixed point

fix fixed point combinator

fact n = F fact n

= (IsZero n) 1

(multiply n (fact (pred n)))

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (3-2)

a fixed-point combinator **fix**

fix F = **F** (**fix F**),

the fixed point **(fix F)** of **F** as **fact**

(fact n) = F (fact n)

define a function **F** of two arguments **f** and **n**:

F f n = (IsZero n) 1 (multiply n (f (pred n)))

F fact n = (IsZero n) 1 (multiply n (fact (pred n)))

fix F = F (fix F),

fix F fixed point

fix fixed point combinator

fact n = F fact n

= (IsZero n) 1

(multiply n (fact (pred n)))

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (4)

(fact n) = F (fact n)

F f n = (IsZero n) 1 (multiply n (f (pred n)))

F fact n = (IsZero n) 1 (multiply n (fact (pred n)))

fact n = F fact n
= (IsZero n) 1 (multiply n (fact (pred n)))

n * fact (n-1)

fix F = F (fix F),

fix F fixed point

fix fixed point combinator

fact n = F fact n
= (IsZero n) 1
(multiply n (fact (pred n)))

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (5)

fact n = F fact n
= (IsZero n) 1 (multiply n (fact (pred n)))

here **(IsZero n)** is a function
that takes two arguments **1 (multiply n (fact (pred n)))**
and returns
its first argument **1** if **n=0**,
otherwise its second argument **(multiply n (f (pred n)))**

pred n evaluates to **n-1**

$$\text{fact } n = \begin{cases} 1 & \text{if } n = 0 \\ n \text{ fact } (n-1) & \text{otherwise.} \end{cases}$$

https://en.wikipedia.org/wiki/Fixed-point_combinator

Recursion (1-1)

recursion:

the definition of a function using the function itself.

A function definition containing itself inside itself, by value,

leads to the whole value being of infinite size.

Other notations which support recursion natively overcome this

by referring to the function definition by name.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (1-2)

Lambda calculus cannot express this:

all functions are anonymous in lambda calculus,

so we can't refer by name to a value which is yet to be defined,

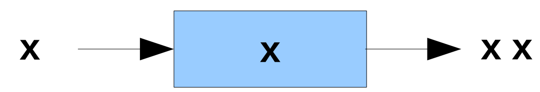
inside the lambda term defining that same value.

however, a lambda expression can receive itself

as its own argument, for example in **(λx.x x) E**.

Here **E** should be an abstraction,

applying its parameter to a value to express recursion.



a function receives itself

as its own argument

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (1-3)

Consider the factorial function **F(n)** recursively defined by

F(n) = 1, if n = 0;   else n * F(n-1).

In the lambda expression which is to represent the function **F(n)**,
a parameter (typically the first one) will be assumed
to receive the lambda expression itself as its value,
so that calling it - applying it to an argument
will amount to recursion.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (2-1)

Thus to achieve *recursion*,
the *intended-as-self-referencing* argument
(called r here) must always be <u>passed</u> to itself **r r**
within the function body, at a call point: **r r (n−1)**

G := λr. λn. (1, if n = 0; else n × (r r (n−1)))

with **r r x = F x = G r x** to hold,

so **r = G** and

F := G G = (λx.x x) G

fix F = F (fix F)

fix F fixed point **fact**
fix fixed point combinator

r r x =
F x =
G r x

r r x =
F x =
G G x

r = G

F = G G = r r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (2-1) – with a self-referencing argument

G is a recursive factorial function

G := λr. λn. (1, if n = 0; else n × (r r (n−1)))

G must have two arguments r x — G r x

in the body of **G**, self-referencing argument **r** must always be passed to **r**, for recursion — r r x

F is the top level function with a single argument x — F x

with **G r x = r r x = F x** to hold — **r = G**

F := G G = (λx.x x) G

G r x =	G G x =
r r x =	r r x =
F x	F x

r = G

F = G G = r r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (3-1)

The self-application achieves replication here,
passing the function's lambda expression
on to the next invocation as an argument value,
making it available to be referenced and called there.

G := λr. λn. (1, if n = 0;   else n × (r r (n−1)))

with **r r x = F x = G r x** to hold, so **r = G**

This solves it but requires re-writing
each recursive call as self-application. **r r (n−1)**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (3-2)

G := λr. λn. (1, if n = 0; else n × (r r (n−1)))

with **r r x = F x = G r x** to hold, so **r = G**

We would like to have a generic solution,

<u>without</u> a need for any <u>re-writes</u>:

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

with **r x = F x = G r x** to hold, so **r = G r =: FIX G** and

r r x =
F x =
G r x

r r x =
F x =
G G x

r = G

F = G G = r r

r x =
F x =
G r x

r x =
F x =
G r x

r = G r

F = G r = r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (2-1) without a self-referencing argument

G is a recursive factorial function

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

G must have two arguments	r x	G r x
in the body of **G**, no self-referencing argument **r**		r x
F is the top level function with a single argument	x	F x

with **r x = F x = G r x** to hold — **r = G r**

G r x =	G r x =
r x =	r x =
F x	G r x

r = G r

F = G r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (3-3)

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

with r x = F x = G r x to hold, so r = G r =: FIX G and

Let r = FIX G, (thus F = FIX G)

(FIX G) = G (FIX G)
r = G r
F = G F

(FIX g) = G (FIX g)
r = g r

F := FIX G where FIX g := (r where r = g r) = g (FIX g)

FIX G = G (FIX G) = (λn. (1, if n = 0; else n × ((FIX G) (n−1))))

r x =
F x =
G r x

r x =
F x =
G r x

r = G r

F = G r = r

fix F = F (fix F)

fix F fixed point fact
fix fixed point combinator

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (4)

FIX G = G (FIX G) = (λn. (1, if n = 0; else n × ((FIX G) (n−1))))

Given a lambda term with first argument representing recursive call (e.g. **G** here), the fixed-point combinator **FIX** will return a self-replicating lambda expression representing the recursive function (here, **F**).

The function does not need to be explicitly passed to itself at any point, for the self-replication is arranged in advance, when it is created, to be done each time it is called.

FIX F = F (FIX F),

FIX F fixed point

FIX fixed point combinator

FIX F = F (FIX F) = fact

(fact) = F (fact)

(fact n) = F (fact n)

F f n = (IsZero n) 1

(multiply n (f (pred n)))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (5)

Thus the original lambda expression (**FIX G**) is
re-created inside itself, at call-point, achieving self-reference.

In fact, there are many possible definitions for this **FIX** operator,
the simplest of them being:

Y := λg.(λx.g (x x)) (λx.g (x x))

Y g = (λx.g (x x)) (λx.g (x x))
= g (λx. (x x)) (λx.g (x x))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (6)

In the lambda calculus, **Y g** is a fixed-point of **g**, as it expands to:

Y g

(λh.(λx.h (x x)) (λx.h (x x))) g

(λx.g (x x)) (λx.g (x x))

g ((λx.g (x x)) (λx.g (x x)))

g (Y g)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (7)

Now, to perform our recursive call to the factorial function,
we would simply call **(Y G) n**, where **n** is the number
we are calculating the factorial of.

Given **n = 4**, for example, this gives:

(Y G) 4
G (Y G) 4
(λr.λn.(1, if n = 0; else n × (r (n−1)))) (Y G) 4
(λn.(1, if n = 0; else n × ((Y G) (n−1)))) 4
1, if 4 = 0; else 4 × ((Y G) (4−1))
4 × (G (Y G) (4−1))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (8)

$4 \times ((\lambda n.(1, \text{if } n = 0; \text{else } n \times ((Y\ G)\ (n-1))))\ (4-1))$

$4 \times (1, \text{if } 3 = 0; \text{else } 3 \times ((Y\ G)\ (3-1)))$

$4 \times (3 \times (G\ (Y\ G)\ (3-1)))$

$4 \times (3 \times ((\lambda n.(1, \text{if } n = 0; \text{else } n \times ((Y\ G)\ (n-1))))\ (3-1)))$

$4 \times (3 \times (1, \text{if } 2 = 0; \text{else } 2 \times ((Y\ G)\ (2-1))))$

$4 \times (3 \times (2 \times (G\ (Y\ G)\ (2-1))))$

$4 \times (3 \times (2 \times ((\lambda n.(1, \text{if } n = 0; \text{else } n \times ((Y\ G)\ (n-1))))\ (2-1))))$

$4 \times (3 \times (2 \times (1, \text{if } 1 = 0; \text{else } 1 \times ((Y\ G)\ (1-1)))))$

$4 \times (3 \times (2 \times (1 \times (G\ (Y\ G)\ (1-1)))))$

$4 \times (3 \times (2 \times (1 \times ((\lambda n.(1, \text{if } n = 0; \text{else } n \times ((Y\ G)\ (n-1))))\ (1-1)))))$

$4 \times (3 \times (2 \times (1 \times (1, \text{if } 0 = 0; \text{else } 0 \times ((Y\ G)\ (0-1))))))$

$4 \times (3 \times (2 \times (1 \times (1))))$

24

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (9)

Every recursively defined function can be seen as a fixed point of some suitably defined function closing over the recursive call with an extra argument, and therefore, using **Y**, every recursively defined function can be expressed as a lambda expression.

In particular, we can now cleanly define the subtraction, multiplication and comparison predicate of natural numbers recursively.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Fix-point combinator (4)

Every recursively defined function can be seen
as a fixed point of some suitably defined function
closing over the recursive call with an extra argument,

and therefore, using **Y**, every recursively defined function
can be expressed as a lambda expression.

In particular, we can now cleanly define
the subtraction, multiplication and comparison predicate
of natural numbers recursively.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (5)

Applied to a function with one variable,

the **Y combinator** usually does not terminate.

More interesting results are obtained

by applying the **Y combinator** to functions of two or more variables.

The additional variables may be used as a counter, or index.

The resulting function behaves like a **while** or a **for** loop

in an imperative language.

Used in this way, the **Y combinator** implements simple recursion.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (6)

In the lambda calculus, it is not possible
to refer to the definition of a function
inside its own body by name.

Recursion though may be achieved
by obtaining the same function passed in as an argument,
and then using that argument to make the recursive call,
instead of using the function's own name,
as is done in languages which do support recursion natively.

The **Y combinator** demonstrates this style of programming.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fix-point combinator (7)

An example implementation of **Y combinator** in two languages is presented below.

```
# Y Combinator in Python

Y=lambda f: (lambda x: f(x(x)))(lambda x: f(x(x)))

Y(Y)
```

https://en.wikipedia.org/wiki/Fixed-point_combinator

Iota combinator (1)

It is possible to define a complete system
using <u>only one</u> (<u>improper</u>) combinator.

An example is Chris Barker's iota combinator,
which can be expressed in terms of **S** and **K** as follows:

ιx = xSK

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Iota combinator (2)

It is possible to reconstruct **S**, **K**, and **I** from the iota combinator.

Applying **ι** to itself gives **ιι = ιSK = SSKK = SK(KK)**
which is functionally equivalent to **I**.

ιx = **xSK**
ιι = **ιSK**
ιS = **SSK**

ιι = (ιS)K = (SSK)K = SK(KK)

ιι y = ιSKy = SK(KK)y = Ky (KK)y = y ➡ **ιι = I**

https://en.wikipedia.org/wiki/SKI_combinator_calculus

ιx = xSK

ιι = ιSK *and* **ιS = SSK**

(ιS)K = (SSK)K

SSKK= SK KK

SKKK = KK KK = K

K K (KK) = K

ιι y = ιSKy = SK(KK)y = Ky (KK)y = y

S K x y = K y (x y) = y

I y = y

Iota combinator (3)

K can be constructed by applying **ι** twice to **I** (**= ιι**)
(which is equivalent to application of **ι** to itself):

ι(ι(ιι)) = ι(ιιSK) = ι(ISK) = ι(SK) = SKSK = K.

ι(ι(ιι)) = ι(ιιSK)
= ι(ISK) ⇐ ιι = I
= ι(SK)
= (SK)SK
= K K SK
= K

ιx = **xSK**
ιι = **ιSK**
ιS = **SSK**

ιι = (ιS)K = (SSK)K = SK(KK)

ιι = ι(SK) = (SK)SK = K K SK = K

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Iota combinator (4)

K can be constructed by applying **ι** twice to **I** (**= ιι**)
(which is equivalent to application of **ι** to itself):

ι(ι(ιι)) = ι(ιιSK) = ι(ISK) = ι(SK) = SKSK = K.

Applying **ι** one more time to **ι(ι(ιι))** gives

ι(ι(ι(ιι))) = ιK = KSK = S

ιx = xSK
ιι = ιSK
ιS = SSK

ιι = (ιS)K = (SSK)K = SK(KK)

ι(ι(ιι)) = ι(ιιSK)
= ι(ISK)
= ι(SK)
= (SK)SK

ιx = xSK
ιι = I
ιιι = K
ιιιι = S

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Improper Combinator

Improper combinators, meaning that they are expressed
in terms of other combinators rather than pure abstractions.

To be precise: in lambda calculus a proper combinator is
an expression of the form $(\lambda.x_1x_2\ldots P(x_1,x_2,\ldots))$,
where $P(x_1,x_2,\ldots)$ only has x_1, x_2 etc. as free variables,
and does not contain any abstractions.

So for example, (λxyz.x(zz)) is a proper combinator,
but (λx.x(λy.y)) is not, because it contains x applied to a lambda term.

https://cs.stackexchange.com/questions/57507/basis-sets-for-combinator-calculus

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf